

The logo of the University of Bordeaux is displayed against a background with a blue diagonal stripe in the top left and a dark grey diagonal stripe in the bottom right. The text 'université' is in a dark brown sans-serif font, with a blue square above the 'i' and a blue 'e' that has a blue underline. Below 'université' is the word 'de' in a smaller dark brown font, followed by 'BORDEAUX' in a large, bold, dark brown sans-serif font.

université
de **BORDEAUX**

Formation CNRS

18 Novembre 2016

Python pour la biologie



Biopython



cgfb

BIOINFORMATIQUE

université
de **BORDEAUX**

Qu'est ce que Biopython ?

- The Biopython Project is an international association of developers of freely available Python tools for computational molecular biology
- The Biopython web site (<http://www.biopython.org>) provides an online resource for modules, scripts, and web links for developers of Python-based software for bioinformatics use and research.
- Basically, the goal of Biopython is to make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and classe

Les fonctionnalités Biopython (1)

- The ability to parse bioinformatics files into Python utilizable data structures, including support for the following formats:
 - › Blast output - both from standalone and WWW Blast
 - › Clustalw
 - › FASTA
 - › GenBank
 - › PubMed and Medline
 - › ExPASy files, like Enzyme and Prosite
 - › SCOP, including “dom” and “lin” files
 - › UniGene
 - › SwissProt

Les fonctionnalités Biopython (2)

- Files in the supported formats can be iterated over record by record or indexed and accessed via a Dictionary interface.
- Code to deal with popular on-line bioinformatics destinations such as:
 - › NCBI { Blast, Entrez and PubMed services
 - › ExPASy { Swiss-Prot and Prosite entries, as well as Prosite searches
- Interfaces to common bioinformatics programs such as:
 - › Standalone Blast from NCBI
 - › Clustalw alignment program
 - › EMBOSS command line tools

Les fonctionnalités Biopython (3)

- Standard sequence class that deals with sequences, ids on sequences, and sequence features.
- Performing common operations on sequences, such as translation, transcription and weight calculations.
- Perform classification of data using k-Nearest Neighbors, Naive Bayes or Support Vector Machines.
- Dealing with alignments, including a standard way to create and deal with substitution matrices.
- Making it easy to split up parallelizable tasks into separate processes.
- GUI-based programs to do basic sequence manipulations, translations, BLASTing, etc.
- Extensive documentation and help with using the modules, including this le, on-line wiki documentation, the web site, and the mailing list.

The Lady Slipper Orchids case

- The Lady Slipper Orchids are in the Orchidaceae family and the Cypripedioideae sub-family and are made up of 5 genera:
- *Cypripedium*
- *Paphiopedilum*
- *Phragmipedium*
- *Selenipedium*
- *Mexipedium*



Working with sequence: The Seq Object

- Most of the time when we think about sequences we have in my mind a string of letters like `AGTACACTGGT`.
- You can create such Seq object with this sequence as follows the “>>>” represents the Python prompt followed by what you would type in
- DON'T FORGET TO USE => `from Bio.Seq import Seq` in your script

```
>>>from Bio.Seq import Seq
>>>my_seq = Seq('AGTACACTGGT')
>>>mv seq
Seq(AGTACACTGGT, Alphabet())
>>> print(my_seq)
AGTACACTGGT
```

```
>>>my_seq.alphabet
Alphabet()
>>>my_seq.complement()
Seq(TCATGTGACCA, Alphabet())
>>>my_seq.reverse_complement()
Seq(ACCAGTGTACT, Alphabet())
```


Sequences et Alphabet: IUPAC Alphabet for DNA, RNA and proteins

- Available alphabets for Biopython are defined in the Bio.Alphabet module.
- IUPAC (<http://www.chem.qmw.ac.uk/iupac/>): Bio.Alphabet.IUPAC
 - › Basic IUPACProtein class
 - › Additional ExtendedIUPACProtein class with Additional elements:
 - "U" (or "Sec" for selenocysteine)
 - "O" (or "Pyl" for pyrrolysine)
 - › Plus the ambiguous symbols:
 - "B" (or "Asx" for asparagine or aspartic acid)
 - "Z" (or "Glx" for glutamine or glutamic acid)
 - "J" (or "Xle" for leucine isoleucine)
 - "X" (or "Xxx" for an unknown amino acid).
 - › IUPACUnambiguousDNA, which provides for just the basic letters
 - › IUPACAmbiguousDNA, which provides for ambiguity letters for every possible situation
 - › ExtendedIUPACDNA, which allows letters for modifiedbase

Sequences et Alphabet (2)

→ Create an ambiguous sequence with the default generic alphabet:

```
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> my_seq.alphabet
Alphabet()
```

→ Specify the alphabet explicitly

```
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

```
>>> from Bio.Alphabet import IUPAC
>>> my_prot = Seq("AGTACACTGGT", IUPAC.protein)
>>> my_prot
Seq('AGTACACTGGT', IUPACProtein())
>>> my_prot.alphabet
IUPACProtein()
```

Sequences act like strings (1)

- Deal with Seq objects as if they were normal Python strings
- For example getting the length, or iterating over the elements:

```
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATC", IUPAC.unambiguous_dna)
>>> for index, letter in enumerate(my_seq):
...     print("%i %s" % (index, letter))

0 G
1 A
2 T
3 C

>>> print(len(my_seq))

4
```

- Access elements of the sequence in the same way as for string

```
>>> print(my_seq[0]) #first letter
G
>>> print(my_seq[2]) #third letter
T
>>> print(my_seq[-1]) #last letter
G
```

Sequences act like strings (2)

- TheSeq object has a “.count()” method, just like a string. Note that this means that like a Python string, this gives a non-overlapping count:

```
>>> "AAAA".count("AA")  
2  
>>> Seq("AAAA").count("AA")  
2
```

- For some biological uses, you may actually want an overlapping count (i.e. 3 in this trivial example). When searching for single letters, this makes no difference:

```
>>> from Bio.Alphabet import IUPAC  
>>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPAC.unambiguous_dna)  
>>> len(my_seq)  
32  
>>> my_seq.count("G")  
9  
>>> 100 * float(my_seq.count('G') + my_seq.count('C')) / len(my_seq)  
46.875
```

Sequences act like strings (3)

- While you could use the above snippet of code to calculate a GC%, note that the Bio.SeqUtils module has several GC functions already built.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> from Bio.SeqUtils import GC
>>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPAC.unambiguous_dna)
>>> GC(my_seq)
```

46.875

- Note that using the Bio.SeqUtils.GC() function should automatically cope with mixed case sequences and the ambiguous nucleotide S which means G or C.
- Also note that just like a normal Python string, the Seq object is in some ways "read-only". If you need to edit your sequence, for example simulating a point mutation, look at the Section 3.12 below which talks about the MutableSeq object.

Slicing a sequence

→ Let's get a slice of the sequence

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq[4:12]
Seq('GATGGGCC', IUPACUnambiguousDNA())
```

→ The new object produced is another Seq object which retains the alphabet information from the original Seq object

→ Get the first, second and third codons positions using “stride” (“::”) :

```
>>> my_seq[0::3]
Seq('GCTGTAGTAAG', IUPACUnambiguousDNA())
>>> my_seq[1::3]
Seq('AGGCATGCATC', IUPACUnambiguousDNA())
>>> my_seq[2::3]
Seq('TAGCTAAGAC', IUPACUnambiguousDNA())
>>> my_seq[::-1] ## Get the reverse sequence
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())
```

Turning Seq objects into strings

- To write to a file, or insert into a database

```
>>> str(my_seq)
'GATCGATGGGCCTATATAGGATCGAAAATCGC'
```

- Calling str() on a Seq object returns the full sequence as a string
- Python does this automatically in the print function

```
>>> print(my_seq)
GATCGATGGGCCTATATAGGATCGAAAATCGC
```

- Also use the Seq object directly with a %s placeholder when using the Python string formatting or interpolation operator (%)

```
>>> fasta_format_string = ">Name\n%s\n" % my_seq
>>> print(fasta_format_string)
>Name
GATCGATGGGCCTATATAGGATCGAAAATCGC
<BLANKLINE>
```

Concatenating or adding sequences

→ Can't add sequences with incompatible alphabets, (protein and DNA)

```
>>> from Bio.Alphabet import IUPAC
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> protein_seq + dna_seq
```

Traceback (most recent call last):

...

TypeError: Incompatible alphabets IUPACProtein() and IUPACUnambiguousDNA()

→ To do this, first give both sequences generic alphabets

```
>>> from Bio.Alphabet import generic_alphabet
>>> protein_seq.alphabet = generic_alphabet
>>> dna_seq.alphabet = generic_alphabet
>>> protein_seq + dna_seq
```

Seq('EVRNAKACGT', Alphabet())

→ Adding a generic nucleotide seq. to an unambiguous IUPAC DNA seq.

```
>>> from Bio.Alphabet import generic_nucleotide
>>> from Bio.Alphabet import IUPAC
>>> nuc_seq = Seq("GATCGATGC", generic_nucleotide)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> nuc_seq + dna_seq
```

Seq(GATCGATGCACGT, NucleotideAlphabet())

Concatenating or adding sequences (2)

→ Many sequences to add together:

```
>>> from Bio.Alphabet import generic_dna
>>> list_of_seqs = [Seq("ACGT", generic_dna), Seq("AACC", generic_dna), Seq("GGTT", generic_dna)]
>>> concatenated = Seq("", generic_dna)
>>> for s in list_of_seqs:
...     concatenated += s
...
>>> concatenated
Seq(ACGTAACCGTT, DNAAlphabet())
```

→ more elegant approach using sum function with its optional start value argument

```
>>> from Bio.Alphabet import generic_dna
>>> list_of_seqs = [Seq("ACGT", generic_dna), Seq("AACC", generic_dna), Seq("GGTT", generic_dna)]
>>> sum(list_of_seqs, Seq("", generic_dna))
Seq(ACGTAACCGTT, DNAAlphabet())
```

Changing case

→ very useful upper and lower methods for changing the case

```
>>> from Bio.Alphabet import generic_dna
>>> dna_seq = Seq("acgtACGT", generic_dna)
>>> dna_seq
```

```
Seq('acgtACGT', DNAAlphabet())
```

```
>>> dna_seq.upper()
Seq(ACGTACGT, DNAAlphabet())
```

```
>>> dna_seq.lower()
Seq(acgtacgt, DNAAlphabet())
```

```
>>> "GTAC" in dna_seq
```

```
False
```

```
>>> "GTAC" in dna_seq.upper()
```

```
True
```

```
>>> from Bio.Alphabet import IUPAC
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> dna_seq
```

```
Seq('ACGT', IUPACUnambiguousDNA())
```

```
>>> dna_seq.lower()
```

```
Seq('acgt', DNAAlphabet())
```

Nucleotide sequences and (reverse) complements

- For nucleotide sequences, you can easily obtain the complement or reverse complement of a Seq object using its built-in methods:

```
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq

Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPACUnambiguousDNA())
```

```
>>> my_seq.complement()

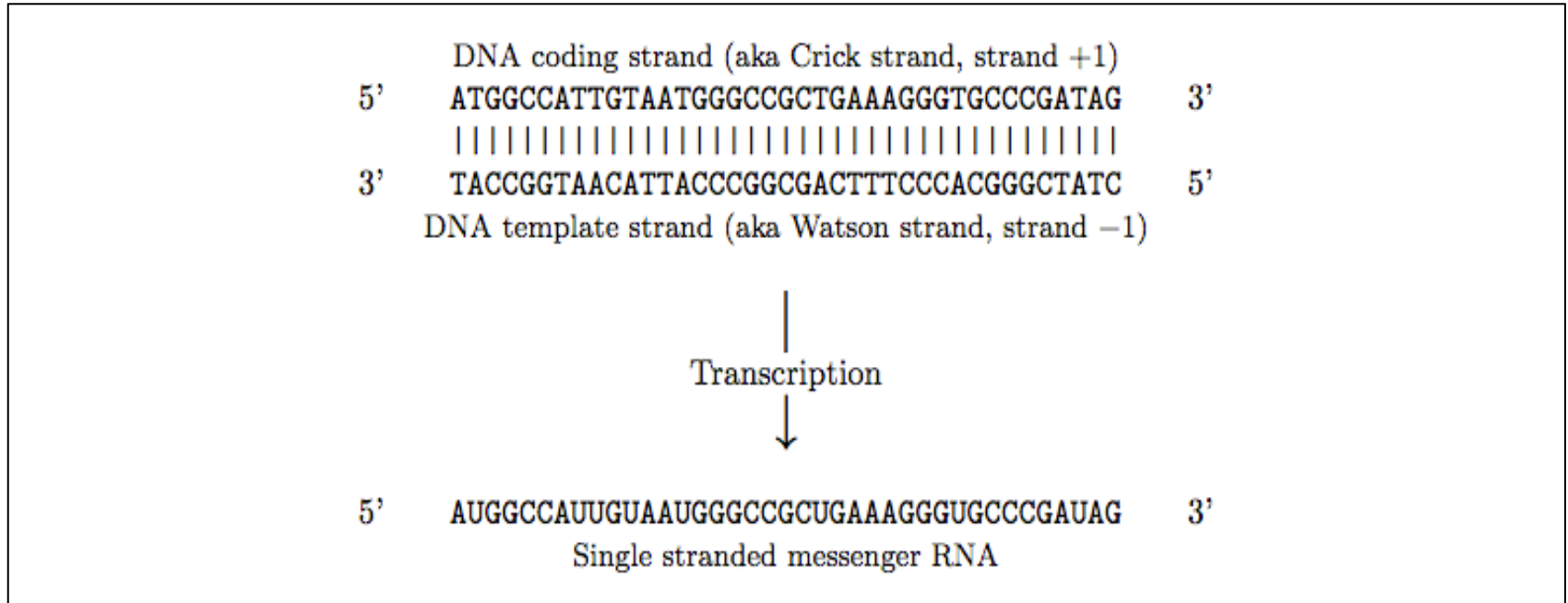
Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG', IUPACUnambiguousDNA())
>>> my_seq.reverse_complement()

Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC', IUPACUnambiguousDNA())
>>> my_seq[::-1]

Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())
```

Transcription

→ Consider the following:



→ The actual biological transcription process works from the template strand, doing a reverse complement (TCAG -> CUGA) to give the mRNA. However, in Biopython and bioinformatics in general, we typically work directly with the coding strand because this means we can get the mRNA sequence just by switching T -> U

Transcription (2)

→ Match the figure above

- › remember by convention nucleotide sequences are normally read from the 5' to 3' direction, while in the figure the template strand is shown reversed.

```
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna

Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())

>>> template_dna = coding_dna.reverse_complement()
>>> template_dna

Seq('CTATCGGGCACCTTTCAGCGGCCCATTAACAATGGCCAT', IUPACUnambiguousDNA())
```

→ Transcribe the coding strand into corresponding mRNA, using Seq object's built in transcribe method (switch T->U and adjust the alphabet)

```
>>> coding_dna

Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())

>>> messenger_rna = coding_dna.transcribe()
>>> messenger_rna

Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Transcription (3) (added in Biopython 1.49)

- Do a true biological transcription starting with the template strand:

```
>>> template_dna.reverse_complement().transcribe()  
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousDNA())
```

- The Seq object also includes a back-transcription method for going from the mRNA to the coding strand of the DNA:

```
>>> from Bio.Alphabet import IUPAC  
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)  
>>> messenger_rna  
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())  
>>> messenger_rna.back_transcribe()  
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

Translation

→ Translate mRNA into the corresponding protein sequence

```
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna

Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.translate()

Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

→ You can also translate directly from the coding strand DNA sequence:

```
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna

Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.translate()

Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

→ Available in Biopython from the NCBI

```
>>> coding_dna.translate(table="Vertebrate Mitochondrial")

Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

→ Using the NCBI table number

```
>>> coding_dna.translate(table=2)

Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

Translation (2)

- Translate nucleotides up to the first in frame stop codon, and then stop

```
>>> coding_dna.translate()  
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

```
>>> coding_dna.translate(to_stop=True)  
Seq('MAIVMGR', IUPACProtein())
```

```
>>> coding_dna.translate(table=2)  
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

```
>>> coding_dna.translate(table=2, to_stop=True) ## the stop codon itself is not translated  
Seq('MAIVMGRWKGAR', IUPACProtein())
```

- complete coding sequence CDS, (e.g. mRNA { after any splicing)
- commences with a start codon, ends with a stop codon, and has no internal in-frame stop codons
- what if your sequence uses a non-standard start codon?
- This happens a lot in bacteria, for example, the gene yaaX in E. coli K12

Translation (3)

```
>>> from Bio.Alphabet import generic_dna
>>> gene = Seq("GTGAAAAGATGCAATCTATCGTACTCGCACTTTCCCTGGTTCTGGTCGCTCCCATGGCA" + \
... "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAATTACAGATAGGCGATCGTGAT" + \
... "AATCGTGGCTATTACTGGGATGGAGGTCACTGGCGCGACCACGGCTGGTGGAAACAACAT" + \
... "TATGAATGGCGAGGCAATCGCTGGCACCTACACGGACCGCCGCCACCGCCGCGCCACCAT" + \
... "AAGAAAGCTCCTCATGATCATCACGGCGGTCATGGTCCAGGCAAACATCACCGCTAA",
... generic_dna)
>>> gene.translate(table="Bacterial")

Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HR*',
HasStopCodon(ExtendedIUPACProtein(), '*'))

>>> gene.translate(table="Bacterial", to_stop=True)

Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

➔ In the bacterial genetic code GTG is a valid start codon, and while it does normally encode Valine, if used as a start codon it should be translated as methionine. This happens if you tell Biopython your sequence is a complete CDS:

```
>>> gene.translate(table="Bacterial", cds=True)

Seq('MKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

Translation Tables

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_name["Standard"]
>>> mito_table = CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
```

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_id[1]
>>> mito_table = CodonTable.unambiguous_dna_by_id[2]
```

```
>>> print(standard_table)
Table 1 Standard, SGC0
```

	T		C		A		G		
T	TTT	F	TCT	S	TAT	Y	TGT	C	T
T	TTC	F	TCC	S	TAC	Y	TGC	C	C
T	TTA	L	TCA	S	TAA	Stop	TGA	Stop	A
T	TTG	L(s)	TCG	S	TAG	Stop	TGG	W	G
C	CTT	L	CCT	P	CAT	H	CGT	R	T
C	CTC	L	CCC	P	CAC	H	CGC	R	C
C	CTA	L	CCA	P	CAA	Q	CGA	R	A
C	CTG	L(s)	CCG	P	CAG	Q	CGG	R	G
A	ATT	I	ACT	T	AAT	N	AGT	S	T
A	ATC	I	ACC	T	AAC	N	AGC	S	C
A	ATA	I	ACA	T	AAA	K	AGA	R	A
A	ATG	M(s)	ACG	T	AAG	K	AGG	R	G
G	GTT	V	GCT	A	GAT	D	GGT	G	T
G	GTC	V	GCC	A	GAC	D	GGC	G	C
G	GTA	V	GCA	A	GAA	E	GGA	G	A
G	GTG	V	GCG	A	GAG	E	GGG	G	G

Translation Tables (2)

```
>>> print(mito_table)
```

Table 2 Vertebrate Mitochondrial, SGC1

	T		C		A		G		
T	TTT	F	TCT	S	TAT	Y	TGT	C	T
T	TTC	F	TCC	S	TAC	Y	TGC	C	C
T	TTA	L	TCA	S	TAA	Stop	TGA	W	A
T	TTG	L	TCG	S	TAG	Stop	TGG	W	G
C	CTT	L	CCT	P	CAT	H	CGT	R	T
C	CTC	L	CCC	P	CAC	H	CGC	R	C
C	CTA	L	CCA	P	CAA	Q	CGA	R	A
C	CTG	L	CCG	P	CAG	Q	CGG	R	G
A	ATT	I(s)	ACT	T	AAT	N	AGT	S	T
A	ATC	I(s)	ACC	T	AAC	N	AGC	S	C
A	ATA	M(s)	ACA	T	AAA	K	AGA	Stop	A
A	ATG	M(s)	ACG	T	AAG	K	AGG	Stop	G
G	GTT	V	GCT	A	GAT	D	GGT	G	T
G	GTC	V	GCC	A	GAC	D	GGC	G	C
G	GTA	V	GCA	A	GAA	E	GGA	G	A
G	GTG	V(s)	GCG	A	GAG	E	GGG	G	G

```
>>> mito_table.stop_codons
```

```
['TAA', 'TAG', 'AGA', 'AGG']
```

```
>>> mito_table.start_codons
```

```
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
```

```
>>> mito_table.forward_table["ACG"]
```

```
'T'
```

Comparing Seq objects

- Meaning of the letters in a sequence are context dependent
- The letter "A" could be part of a DNA, RNA or protein sequence.
- Comparing two Seq objects could mean considering both the sequence strings and the alphabets
- Compare the sequences as string:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.ambiguous_dna)
>>> str(seq1) == str(seq2)
True

>>> str(seq1) == str(seq1)
True
```

- Sequence comparison only looks at the sequence, ignoring alphabet

```
>>> seq1 == seq2
True
>>> seq1 == "ACGT"
True
```

- Note if you compare sequences with incompatible alphabets (e.g. DNA vs RNA, or nucleotide versus protein), then you will get a warning but for the comparison itself only the string of letters in the sequence is used:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna, generic_protein
>>> dna_seq = Seq("ACGT", generic_dna)
>>> prot_seq = Seq("`ACGT", generic_protein)
>>> dna_seq == prot_seq
BiopythonWarning: Incompatible alphabets DNAAlphabet() and ProteinAlphabet()
True
```

- **WARNING:** Older versions of Biopython instead used to check if the Seq objects were the same object in memory.
- Important if you need to support scripts on both old and new versions of Biopython.
- Make the comparison explicit by wrapping your sequence objects with either `str(...)` for string based comparison or `id(...)` for object instance based comparison.

MutableSeq objects

→ The Seq object is “read only”, or in Python terminology, immutable

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
```

→ Observe what happens if you try to edit the sequence:

```
>>> my_seq[5] = "G »
Traceback (most recent call last):
TypeError: 'Seq' object does not support item assignment
```

→ However, you can convert it into a mutable sequence (a MutableSeq object) and do pretty much anything you want with it:

```
>>> mutable_seq = my_seq.tomutable()
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
```

→ Alternatively, you can create a MutableSeq object directly from a string:

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
```

→ Either way will give you a sequence object which can be changed:

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq[5] = "C"
>>> mutable_seq
MutableSeq('GCCATCGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.remove("T")
>>> mutable_seq
MutableSeq('GCCACGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.reverse()
>>> mutable_seq
MutableSeq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

→ Note that unlike the Seq object, the MutableSeq object's methods like `reverse_complement()` and `reverse()` act in-situ!

```
>>> new_seq = mutable_seq.toseq()
>>> new_seq
Seq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

UnknowSeq objects

- Subclass of the basic Seq object
- Represent a sequence where we know the length, but not the actual letters making it up.

```
>>> from Bio.Seq import UnknownSeq
>>> unk = UnknownSeq(20)
>>> unk
UnknownSeq(20, alphabet = Alphabet(), character = '?')
>>> print(unk)
????????????????????
>>> len(unk)
20
```

- Specify an alphabet, meaning for nucleotide sequences the letter defaults to “N” and for proteins “X”, rather than just “?”

```
>>> from Bio.Seq import UnknownSeq
>>> from Bio.Alphabet import IUPAC
>>> unk_dna = UnknownSeq(20, alphabet=IUPAC.ambiguous_dna)
>>> unk_dna
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> print(unk_dna)
NNNNNNNNNNNNNNNNNNNNNN
```